# Tolerance As An Example[1]

Padmanabhan Krishnan
Department of Computer Science
University of Canterbury
Private Bag 4800
Christchurch 1, New Zealand
E-Mail: paddy@cosc.canterbury.ac.nz

**Abstract**

In this paper we show how the action semantics framework can be used to describe a particular implementation of fault-tolerant systems. We also define a notion of simulation which can be the basis for relating a fault-tolerant implementation to an abstract (non-faulty, and non-fault tolerant) specification. The aim of the paper is to illustrate that good software engineering techniques can be applied to semantic descriptions. Issues such as modularity, extensibility of the semantic descriptions is illustrated.

## 1 Introduction

Software reuse is considered to be one of the principal areas from which productivity gains is expected. Various researchers [1, 2]. have shown that there are various aspects to reuse and it is important to realise the scope and the effect of reuse. Just as reuse is important for software development, reuse can also be crucial in the development of formal specifications. In general, development and maintenance of formal specifications is no different from the development and maintenance of software systems.

In software systems, if one has to extend the domain of application, one does not rewrite the software from scratch. However, as far as we are aware, no software engineering principles are applied to the development of formal specifications. The traditional role of formal specification is as a fixed entity that guides issues such as verification, validation and implementation. The development of formal specifications itself could be error prone and it is essential to apply something like the waterfall model [15] to it. Furthermore, if a formal specification is to have a lasting value, it should be easy to update it to obtain extensions.

The aim of this paper to show that action semantics [9] is a good choice for formal specifications as it exhibits modularity which enables the reuse of parts of a specification. The scalability of action descriptions has been shown in [11] where the addition of concurrency had minimal influence on the semantic definitions used to describe sequential computation. The underlying notation and its semantics itself required no change. The addition of interrupts to the notation was essential to model certain aspects of fairness. This required a change to the operational semantics of the notation [6, 5]. But overall the nature of the changes were simple and the changes themselves were fairly small.

In this paper we focus on the semantics of a simple language in which certain types of fault-tolerant systems can be expressed. With safety critical systems gaining in importance [3], the software engineering aspects of formal specifications is becoming more relevant.

[1]Work in Progress: Supported by UoC Grant 1787123

We begin by considering a semantic description for a simple sequential language. An extension to specify the semantics of faults is considered which is followed by an extension to specify the semantics of fault-tolerance.

## 2 Notational Details

Action semantics uses a notation in which the semantics of realistic languages like Pascal or Ada can be defined. Such semantic descriptions are compositional (as in traditional denotational semantics) where the co-domain of the semantic functions contains actions (instead of $\lambda$-terms or higher order functions). Actions are objects that have an operational intuition and indeed the semantics of actions is described using the SOS approach [9][pages 278,295]. The SOS defined the actions induces an operational semantics for the language being developed and this is used to develop compiler generators [13, 14, 12].

The entire notation is based on a set of primitive actions and various combinators to create more complex actions. The application of the operational semantics for actions results in the processing of information. Depending on the type of information being processed actions are categorised into various facets.

Some predefined data notation, which includes numbers, characters, sets, tuples, is provided. This can be extended to define any data type required by the semantics. Certain classes of values which depend on the state of the computation (called yielders) are also identified. The yielders are evaluated to get a specific value of the appropriate type.

The following tables along with their intuitive descriptions summarise the various facets along with their primitive actions and a few combinators. The reader is referred to [9] (pages 261-277) for details.

| Actions/Yielders | Combinators |
|---|---|
| complete, diverge, escape<br>fail, commit, unfold | and then, or<br>trap, unfolding |

Table 1: Basic Actions

The action complete always terminates, while diverge never terminates. The action fail indicates abortive termination and is used to abandon the current alternative. The action commit corresponds to cutting away all alternatives, while the action escape corresponds to raising an exception. The combinator and then corresponds to sequential performance while the combinator and performs two actions with arbitrary interleaving. The combinator or represents non-deterministic choice. An alternative to the chosen action is performed when the chosen action fails (unless a commit has been performed). The combinator trap is used to handle exceptions raised using escape. The combinator unfolding along with the basic action unfold specifies iteration. unfolding A performs A, but when the action unfold is encountered in A, the action A is performed.

The action give D yields the datum D while the action give D#$n$ yields the $n$'th component of the tuple represented by D. The action regive regenerates any data given to it and is useful to make copies of the given data. The action choose S gives an element of the data of sort S while check D completes if D is the boolean true; fails otherwise. The principal functional combinator is then. $A_1$ then $A_2$ corresponds to functional composition, i.e., $A_2$ is given the data produced by $A_1$.

| Actions/Yielders | Combinators |
|---|---|
| give, choose, regive check | then |

Table 2: Functional Actions

| Actions/Yielders | Combinators |
|---|---|
| bind to , rebind, produce, bound to current bindings | moreover, hence, before |

Table 3: Declarative Actions

The declarative actions process scoped information and associate tokens (identifiers in the semantic domain) with values. The action bind T to D, which produces a binding of token T to datum D, rebind which reproduces all the bindings it received and produce D which converts the data item D into a binding. Information from the current bindings can be extracted by the S bound to T returns the datum (if it is of sort S) bound to the token T. The data specification current bindings converts the entire set of bindings into data. This combined with produce permits the manipulation of bindings as data and reconverting data into bindings.

The action $A_1$ moreover $A_2$ corresponds to letting bindings produced by $A_2$ override those produced by $A_1$, i.e., bindings produced by $A_2$ have a higher precedence. The action furthermore A is similar and produces the same bindings as A along with any received bindings that is not overridden by A. The action $A_1$ hence $A_2$ restricts the bindings received by $A_2$ to those produced by $A_1$ and bindings produce by $A_2$ is propagated. This limits the scope of bindings produced by A1 unless A2 reproduces them.

| Actions/Yielders |
|---|
| store, allocate, stored in deallocate |

Table 4: Imperative Actions

The imperative actions deal with storage, consisting of individual cells, which is stable information. The action store $D_1$ in $D_2$ stores the datum $D_1$ in cell $D_2$ while allocate D corresponds to the allocation of a cell of sort D while the action deallocate D destroys the allocation of cell corresponding to D. Data of sort S that is stored in a cell D can be extracted by ( S stored in D).

In many cases it is necessary to treat actions as data. For example, binding the body of a procedure to an identifier, the action representing the body needs to be treated as data. An abstraction is a data type that incorporates an action. Abstractions are created using the constructor abstraction of. References to transient data in an action is not evaluated when the abstraction is created. Transient information (i.e., parameters) can be given to the abstraction by application Abs to D, where the data D is supplied to Abs. Similarly bindings can be supplied to abstraction Abs using closure Abs. Actions converted into abstractions can be performed using the action enact. For example, the abstraction Abs is executed by enact Abs.

| Actions/Yielders |
| --- |
| enact, application to , closure, abstraction of |

Table 5: Reflective Actions

The action notation also supports concurrency. Concurrent behaviour is represented by agents which evolve asynchronously. The agents can communicate via message passing which can be used to synchronise agents. This concludes our brief overview of the action notation. A reader who is interested in the more technical aspects of the notation is referred to [9] where the operational semantics for the notation and a number of algebraic laws that the actions satisfy are developed.

## 2.1 A Simple Sequential Language

In this section we describe the syntax and semantics of a simple language. We will assume that variables hold only numbers and are created statically. The purpose of this language is only to illustrate the use of the various actions/combinators. We will extend this language to address issues of fault-tolerance.

Towards defining the semantics of this language, we define four semantic functions. The first, establish, creates the necessary storage for all the variables that hold numbers. The second, evaluate describes the evaluation of expressions while the third, execute, defines the semantics of execution, i.e., the flow of control and state changes. The final equation, run, defines the semantics of programs which at first establishes bindings and then executes the program. The formal definitions are given below.

Id = ⟦ letter$^+$ ⟧

Expr = Id | ⟦ Expr "+" Expr ⟧ | ⟦ Expr "=" Expr ⟧

St = ⟦ Id ":= Expr ⟧ | ⟦ "if" Expr "then" St "else" St ⟧ |
⟦ "while" Expr "do" St ⟧ | ⟦ St ";" St ⟧

Pgm = St | ⟦ Id ":" Pgm ⟧

establish :: Id → action

establish I:Id = allocate a num-cell then bind it to the token of I

evaluate :: Expr → action

evaluate I:Id = give the contents of (the cell bound to token of I)

evaluate ⟦ E1 "+" E2 ⟧ = | evaluate E1 and evaluate E2
then
| give the sum(number #1, number #2)

evaluate ⟦ E1 "=" E2 ⟧ = | evaluate E1 and evaluate E2
then
| give same(number #1, number #2)

execute :: St → action

```
execute ⟦I:Id "::=" E:Expr⟧ = evaluate E then assign the value (to the datum bound to token of I)

execute ⟦ "if" E:Expr "then" S1:St "else" S2:St ⟧ = evaluate E then
                                                      | check (it is true) and then execute S1
                                                    or
                                                      | check (it is false) and then execute S2

execute ⟦ "while" E:Expr "do" S:St ⟧ = unfolding
                                       | evaluate E then
                                       | | check (it is true) and then
                                       | | | execute S1 and then unfold
                                       | or
                                       | | check (it is false)

execute ⟦S1:St ";" S2:St ⟧ = execute S1 and then execute S2

run :: Pgm → action

run S:St = execute St

run ⟦ I:Id ":" P:Pgm⟧ = establish I moreover run P
```

The semantic equations are quite straightforward. More details can be obtained from the action semantics tutorial [10].

## 3 Fault Specification

We extend the simple language to include faults. We consider two types of faults. They are garbling of state (i.e, values associated with variables) and crash failure (i.e., a cell becomes inaccessible). Following [4], we model faults as 'normal processing' which operates in asynchronous conjunction with the rest of the program. Thus the system has no control over when (if at all) the faults occur.

The formal extension to the grammar is given below.

```
Failure  = ⟦ "corrupt" Id ⟧ | ⟦ "fail" Id ⟧ | ⟨Failure+⟩

FPgm     = ⟦ St "|" Failure⟧ | ⟦ Id ":" FPgm ⟧
```

The inclusion of Failure is a pure addition to the original grammar while the old program had to be extended to include potential faults. This requires us to add a new semantic function, fexecute, for Failure and alter the semantics of run to obtain frun which uses the semantics of fexecute.

As this point we have not yet addressed the issue of fault-tolerance. All we have done is to specify the semantics of faults.

```
fexecute ⟦ "corrupt" I:Id ⟧ = | | give the datum bound to token of I
                              | and
                              | | choose a number
                            then
                              | assign the number #2 to the datum #1

fexecute ⟦ "fail" I:Id ⟧ = | give the datum bound to token of I
                         then
                           | deallocate it
```

fexecute ⟨F1:Failure F2:Failure$^+$⟩ = fexecute F1 and fexecute F2

frun :: FPgm → action

frun ⟦ S:St "|" F:Failure⟧ = execute S and fexecute F

frun ⟦ I:Id ":" P:FPgm⟧ = establish I moreover frun P

The semantic definitions are as expected with all the faults operating in asynchronous fashion. As the effect of the faults depends on the bindings of identifiers, the use of the combinator moreover after establish and and between execute and fexecute ensures that the statements and the faults get the same bindings.

# 4 Fault Tolerance

In this section we add features which are useful in building fault-tolerant systems. We use replication of a cell to withstand data corruption and failure. The degree of replication is specified by the programmer.

Protect = ⟦ "copies" Id Expr ⟧ **|** ⟦ Protect$^+$ ⟧

p-establish :: Protect → action

```
p-establish ⟦ "copies" I:Id E: Expr ⟧ = evaluate E then
                                        | replicate (the given number#1) then
                                        | | bind them to the token of I
```

replicate 1 = allocate a m-cell

replicate n = allocate a r-cell and (replicate (n-1))

The degree of replication is specified by an expression. Instead of using num-cells we now use m-cell and r-cells. The reason for the two types of cells will become clear later when we relate the extended semantics to the original one. Intuitively, we use the m-cell as an anchor to relate it to the num-cell.

As we have changed the structure of the bindings of identifiers, the semantic equations evaluate and execute need to be changed. These are specified below.

```
evaluate I:Id = give the datum bound to token of I then
                vote-value them

execute ⟦I:Id ":=" E:Expr⟧ = | evaluate E and give the datum bound to token of I
                             then
                             | assign-forall (first of them) to (the rest of them)
```

We leave the exact specification of voted-value and assign-forall open. The intuition is that in voted-value all the legal cells are inspected. The values are then accumulated and a voting strategy applied to them. For example, one can adopt majority voting or an average value voting. The process of inspecting legal cells is given below.

```
inspect d = | choose (d & current-storage) and then give the contents of d
            or
            | choose (disjoint-union (d, current-storage)) and then complete
```

If a cell has been deallocated due to failure, no value is returned. Notice that we do not change the bindings as we do not have any technique of passing on the new bindings to the other statements. If a change in bindings has to affect most of the actions, it is almost essential to commit them to stable storage.

The intuition behind assign-forall is similar in that it assigns the same value to all the currently legal cells.

Due to a change in the representation of the cell, the semantic equations describing the meaning of faults also need to be changed.

```
fexecute ⟦"corrupt" I:Id⟧ = | | give the datum bound to I then (select-one #1)
                            | and
                            | | choose a number
                            then
                            | assign the number #2 to the datum #1

fexecute ⟦ "fail" I:Id ⟧ = | give the datum bound to I then (select-one #1)
                           then
                           | deallocate it
```

The nature of the change is similar to the changes in execute etc. We leave the exact semantics of select-one unspecified. The intuitive behaviour of select-one is to non-deterministically select a r-cell which will then be corrupted by the assignment. Note this effect could also be achieved by changing the semantics of assign and deallocate.

### 4.1 Discussion of Changes

The addition of faults and fault-tolerant aspects has altered a few of the original semantic equations. However the changes were very localised. More specifically they were only to equations (and furthermore restricted to parts of equations) that explicitly dealt with the representation of identifiers, values and the communication.

If the original semantic equations could have been written in a style which used abstract data types (e.g., never exposed the structure of a binding), the changes would have been to the semantic entities only. Following an abstract data type prescription for semantics results in overly verbose descriptions and in most situations, the semantics are not drastically altered very often. What we presented as the original definition is a realistic expectation of an action description. The price paid to extend the original description to cater to fault tolerance is not very high considering the radical nature of the change. For a large language many equations will not be altered. In our toy example, we have seen that the semantics of statement sequencing, while loops etc. required no change.

## 5 Simulation Relation

A notion of bisimulation and testing equivalences for actions is defined. These relations based on the notion of commitments which are either messages or changes to the store. The configurations for each agent are not just actions but is a combination of actions with their given information (such as transients, bindings) operating on the local information of store and messages. The details of this is specified in [9](pages 261-295). Here we present a slightly simplified view.

state = (Acting, local-info)

local-info = (storage, buffer)

Acting = (Action, data, bindings) | (Acting In-fix Acting) | (Pre-fix Acting)

Acting represents actions operating on their given information while local-info contains the state of the store, the incoming message buffer and the identity of the agent.

The operational semantics is defined by a function stepped. If an action associated with a state s can be performed, stepped s yields the states s can evolve into along with the communications generated by the performance. An auxiliary function simplified is defined which handles the propagation of transients and bindings and termination details. For example, state ⟦ completed and A ⟧ is simplified to state ⟦ A ⟧.

(1) stepped _ :: state → (state, commitment)

(2) commitment = list of [communication] | uncommitted

The empty list is used to indicate changes to the store. This is sufficient as the exact changes are recorded in the storage component of local-info. All other transitions, such as creating a transient or a binding, reading the store etc., are labelled by uncommitted.

The following rules help to define the semantics for and.

(1) stepped (state A1 (s h)) :- (state A1' (s' h')) c' ⇒
stepped (state ⟦ A1 "and" A2 ⟧ (s h)) :- simplified (state ⟦ A1' "and" A2 ⟧ (s' h')) c'

(2) stepped (state A2 (s h)) :- (state A2' (s' h')) c' ⇒
stepped (state ⟦ A1 "and" A2 ⟧ (s h)) :- simplified (state ⟦ A1 "and" A2' ⟧ (s' h')) c'

Recall that the and combinator defines the interleaved execution of two actions. The first rule states that if the state A1 (s h) can make a transition to the state A1' (s' h') c', ⟦ A1 "and" A2 ⟧(s h) can make a transition to ⟦ A1' "and" A2 ⟧ (s' h') c'. The second rule specifies the progress of A2.

Based on the above definition a state transition relation $\longrightarrow$ is defined as follows: s $\xrightarrow{c}$ s′ iff (s′,$c$) : stepped s. An observable transition $\Longrightarrow$ is defined as $\overset{c}{\Longrightarrow} = (\overset{\text{uncommitted}}{\longrightarrow})^{*} \xrightarrow{c} (\overset{\text{uncommitted}}{\longrightarrow})^{*}$

**Definition: 1** *Two actions A1 and A2 are equivalent iff forall local information l*

*(A1 l)* $\xrightarrow{c}$ *(A1′,l′) then (A2 l)* $\overset{c}{\Longrightarrow}$ *(A2′, l′) and (A1′, l′) is equivalent to (A2′, l′) and*

*(A2 l)* $\xrightarrow{c}$ *(A2′,l′) then (A1 l)* $\overset{c}{\Longrightarrow}$ *(A1′, l′) and (A1′, l′) is equivalent to (A2′, l′)*

The above definition is similar to the observational equivalence defined in [7].

This definition is too strong for our purposes. It requires an exact match of all state changes. This is clearly not the case in the fault tolerant setting. Due to replication of cells, a single assignment is translated into multiple assignments. However, we would still like to relate the extended semantics (faults and fault tolerance) with the original semantics. Towards this we introduce a notion of simulation which ignores certain aspects of the behaviour.

### 5.1 Derived Relation

As we would like to relate the original 'perfect world' semantics to the 'fault-tolerant' semantics, what is necessary is a relationship between the data structures used in the two semantics. Influenced by the work described in [8], we introduce a function similar to the notion of abstraction invariant. This function provides a map between the various sorts used in the semantics.

As we have introduced replication, a single cell access has been translated to multiple cell accesses. In order to use the ideas behind observational equivalence, we have rename all the extra accesses to uncommitted. The derived relation is indexed by such a function and the usual definition of observational equivalence can be used.

**Definition: 2** *Two actions A1 and A2 are equivalent under an abstraction function F iff for all local information l*

- *(A1 l)* $\xrightarrow{c}$ *(A1',l') and F(c)* $\neq$ uncommitted *then (A2 l)* $\overset{cs}{\Longrightarrow}$ *(A2', l') where*

  $cs = c_1, c_2, \ldots c_n$ *with* $F(c_i) = c$ *and for all other* $j$, $F(c_j)$ = uncommitted

  *(A1', l') is equivalent to (A2', l')*

- *(A1 l)* $\xrightarrow{c}$ *(A1',l') and F(c) =* uncommitted *then either*

  *[(A1 l)* $\overset{\text{uncommitted}^*}{\Longrightarrow}$ *(A1', l') and (A1', l') is equivalent to (A2', l')] or*

  *[(A1, l) is equivalent to (A2', l')]*

- *Similarly for A2*

In the above definition, the effect of $F$ is to internalise the actions that $F$ maps to uncommitted.

This definition can be applied to our example to show that the semantics of a given program is identical to another given certain fault assumptions and fault-tolerant techniques. For example, to withstand one fault, a triple replication with majority voting suffices.

By defining $F$ to be such that $F$(m-cell) = num-cell and $F$(r-cell) = uncommitted, one can show that the fault tolerant semantics can be related to the perfect semantics. This is not always true, as the degree of replication may not be enough to withstand the number of faults injected into the system.

## 6 Conclusions and Future Work

Here we have added aspects of fault tolerance and again the good pragmatic features of action semantics have been demonstrated. We have also defined a general framework in which various extensions to an existing semantics can be related to the original one. This forms the basis for proofs of correctness of extensions and is under further investigation.

The technique we have outline can be adapted to suit other situations. For example, we can translate an array assignment (which can be a single assignment to a complex cell) into a series of assignments to individual simpler cells. By making one of the cells in the collection as a distinguished one, one can relate the sequence of assignments to the single array assignment. By increasing the domain of the function $F$ to include messages, it is possible to specify various parallel implementations of languages.

## References


[1] B. Barnes and T. Bollinger. Making reuse cost-effective. *IEEE Software*, pages 13–24, January 1991.

[2] V. Basili and H. Rombach. Support for comprehensive reuse. *The IEE Software Engineering Journal*, 6:303–316, Sept 1991.

[3] J. Bowen. Formal methods and safety-critical standards. *IEEE Computer*, 27(8):68–71, August 1994.

[4] F. Cristian. A Rigorous Approach to Fault-Tolerant Programming. *IEEE Transactions on Software Engineering*, 11(1):23–31, 1985.

[5] P. Krishnan. Specification of Systems with Interrupts. *Journal of Systems and Software: Special Issue on Applying Specification, Verification and Validation Techniques to Industrial Software Systems*, 21(3):291–304, June 1993.

[6] P. Krishnan and P. D. Mosses. Specifying Asynchronous Transfer of Control. In J. Vytopil, editor, *Proceedings of the Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems: LNCS 571*, pages 291–306, Nijmegen, Netherlands, January 1992. Springer Verlag.

[7] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[8] C. C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.

[9] P. D. Mosses. *Action Semantics*. Number 26 in Tracts in Theoretical Computer Science. Cambridge University Press, August 1992.

[10] P. D. Mosses. A Tutorial on Action Semantics. In *Proceedings of FME*, Barcelona, Spain, September 1994.

[11] P. D. Mosses and M. Musicante. An Action Semantics for ML Concurrency Primitives. In *Proceedings of FME*, Barcelona, Spain, September 1994.

[12] Peter Ørbæk. OASIS: An Optimizing Action-based Compiler Generator. In Peter Fritzon, editor, *Proceedings of the 1994 Conference on Compiler Construction , Edinburgh*, volume 786 of LNCS, pages 1–15. Springer-Verlag, April 1994.

[13] J. Palsberg. A Provably Correct Compiler Generator. In *European Symposium On Programming (ESOP): LNCS 582*, pages 418–434, Rennes, France, February 1992. Springer-Verlag.

[14] J. Palsberg. An Automatically Generated and Provably Correct Compiler for a Subset of Ada. In *Fourth IEEE International Conference on Computer Languages*, San Fransisco, California, April 1992.

[15] Ian Sommerville. *Software Engineering*. Addison Wesley, third edition, 1989.